Standardized Intermediate Representation for Fortran, Java, C and C++ Programs

APART[*]Technical Report
Workpackage 1

http://www.fz-juelich.de/apart

Aurora Technical Report

Clovis Seragiotto, Jr., Hong-Linh Truong
Institute for Software Science
University of Vienna
{*clovis, truong*}*@par.univie.ac.at*

Thomas Fahringer
Institute for Computer Science
University of Innsbruck
Thomas.Fahringer@uibk.ac.at

Bernd Mohr
Central Institute for Applied Mathematics
Forschungszentrum Jülich GmbH
B.Mohr@fz-juelich.de

Michael Gerndt, Tianchao Li
Institut für Informatik, LRR
Technische Universität München
{*gerndt, lit*}*@in.tum.de*

Abstract

When building a performance analysis tool, one usually needs to select an instrumentation engine for a specific programming language. Instrumentation engines commonly traverse an internal, mostly non-portable program representation in order to insert probes that collect performance information during execution of an instrumented program. Performance analysis tools are built on non-standardized instrumentation engines, which forces performance tools to develop separate interfaces to instrumentation engines and adjust to different intermediate program representations, one for every different instrumentation engine. In the context of the Apart working group, we designed a unique intermediate program representation named SIR as an XML-language, which we propose as a standard to be used by a large variety of performance analysis tools. The basic idea of SIR is to provide a standardized intermediate program representation that must be supported by a large variety of instrumentation engines covering the most important programming languages. SIR has been defined thus far for Fortran, C, C++, and Java thus we demonstrated that SIR comprises both procedural and object-oriented programming languages. Through a SIR, performance analysis tools no longer have to adjust to different program representations for every different instrumentation engine. Moreover, in combination with a standardized instrumentation request language, a comprehensive interface between performance analysis tools and instrumentation engines is described.

1 Introduction

Many performance tools rely on performance information (overheads, trace files, etc.) commonly obtained by statically instrumenting the source code or dynamically instrumenting an executable. Static instrumentation frequently imposes compiler dependencies, whereas dynamic instrumentation may depend on symbol table information and is also very much hardware

[*]The IST Working Group on *Automatic Performance Analysis: Real Tools* is funded under Contract no. IST-2000-28077

specific. In the worst case performance tool developers have to build separate instrumentation engines for every different programming language, a tedious and time consuming effort.

In this document we propose a **S**tandardized **I**ntermediate **R**epresentation as an interface between instrumentation engines and higher-level performance tools. The SIR is intended to be an abstract representation for procedural and object-oriented programs. Basically, a SIR contains information about statements and directive types (e.g. OpenMP) with very little details on their structures. Higher-level performance tools commonly only need to know the type of a statement in order to make a decision about specific instrumentation requests or performance analysis.

Moreover, higher-level performance tools can request the generation of a SIR for arbitrary source programs. The performance tool can then traverse the SIR and request the instrumentation of specific code regions. Important is that the generation of a SIR based on a specific input program as well as the actual instrumentation is done by an external tool. On the basis of the SIR, higher-level performance tools are provided with a high-level and portable instrumentation/profiling/monitoring interface for a broad variety of programming languages without dealing with low-level details such as instrumentation, tracing, etc.

In this document we describe the format of a SIR for Fortran 95, Java, C and C++ programs applicable not only to source codes but also to binaries generated from a program written in one of these languages. As extensive support already exists to traverse XML documents, we chose to define the SIR based on XML.

1.1 Term Definitions

An *XML document* is a well-formed data object according to the XML specification [1]. An XML document contains at least one *element*, and each element may have a set of *attributes* and may be nested within other elements. The (unique) element in the XML document that is not nested is called the *root element*. For instance, in

```
<staff>
    <employee matr="B001" name="John Doe" marriedTo="A003"/>
    <employee matr="B002" name="John Smith"/>
    <employee matr="A003" name="Jane Doe" marriedTo="B001"/>
</staff>
```

<*staff*> and <*employee*> are elements, while *matr*, *name* and *marriedTo* are attributes of the element <*employee*>.

A *DTD* (Document Type Definition) is a set of markup declarations that defines the grammar for a class of XML documents. An XML document that has an associated DTD and complies with it is said to be *valid*. For example, the previous example is valid according to the following DTD:

```
<!ELEMENT staff (employee)+> <!-- meaning: a staff element may contain one -->
                             <!-- or more employee elements -->
<!ELEMENT employee EMPTY> <!-- meaning: an employee element may not contain -->
                            <!-- any text or nested element -->
<!ATTLIST employee          <!-- meaning: an employee element: -->
  matr ID #REQUIRED         <!--   must have the attribute matr, a string not -->
                            <!--    used as the value of any other ID-attribute -->
                            <!--    in the same XML document -->
  name CDATA #REQUIRED      <!--   must have the attribute name, a string -->
  marriedTo IDREF #IMPLIED  <!--   may have the attribute marriedTo, a string -->
                            <!--     with the same value of any other -->
                            <!--     ID-attribute in the same XML document-->
```

In the following, DTD will be used to define the *grammar* of SIR documents, but note that DTD cannot describe all semantic rules and restrictions specificated in this document.

DTDs have a limited datatype capability and a different syntax from XML documents; this motivated the development of *XML schemas*, to which the DTD grammar can be converted using the rules in Section 4.

2 SIR Description

A SIR is an XML document representing a Fortran 95, Java, C or C++ program (referred to simply as *input program* in the rest of this document). A valid SIR may contain several types of elements, the most important of which are *sir*, *unit*, and *codeRegion*. All the elements are described in detail in this section, which also gives the markup declarations (element type declarations and attribute-list declarations) that must compose the DTD describing the syntax of SIRs.

A tool that generates SIRs does not need to represent all elements described in this specification in order to be SIR compliant; nevertheless, the tool must document that, if the generated SIR does not contain a certain element or attribute, this is not because the element is absent in the input program, but because the tool chose to ignore it.

2.1 The Element sir

The root of any SIR is given by a *sir* element. A *sir* element must specify:

- the ”main” language the input program is written in (for instance, if a Java program uses native C functions, the language must be Java, not C);
- at least one group (for instance, a class; see Section 2.2) or a program unit (for instance, a function; see Section 2.3).

Moreover, if it is known that the program communicates with other programs (processes or threads) by sending and receiving messages, the *messagePassing* attribute may be specified with the value *true* (default: *false*); similarly, if communication is done (also) through shared memory, the *sharedMemory* attribute may be specified with the value *true* (default: *false*).

The markup declarations representing these requirements are given below:

```
<!ELEMENT sir (variable*, (group|unit)+)>
<!ATTLIST sir
  language (fortran|java|c|cpp) #REQUIRED
  messagePassing (true|false) #IMPLIED
  sharedMemory (true|false) #IMPLIED>
```

2.2 The Elements group and inheritance

The *group* element is used to represent an organizational, non-executable unit:

- modules in Fortran;
- packages, classes, interfaces, array types, and enums in Java;
- namespaces and classes, as well as structs and unions that define methods, in C++.

Any *group* element must specify:

- the type of the group it represents (with the attribute *type*);
- a unique identifier (with the attribute *id*).

The type used for Java arrays and enums and for C++ structs and unions is *class*, while the type used for C++ namespaces is *package*.

A *group* element may also specify the name of the group it represents (using the attribute *name*) and the internal name the compiler assigned to the represented group (with the attribute *internal*). In C++, aliases of namespaces are ignored, as well as any alias for a class, struct or union name created with *typedef*. This rule holds also for struct and union names in C.

A *group* element contains also zero or more *group* elements and zero or more *unit* elements.

The declaration of variables (or fields) is represented with the element *variable*, described in Section 2.5.

When representing a class or interface, a *group* element may specify superclasses and superinterfaces using the element *inheritance*; this element accepts either the identifier or the name of a superclass or superinterface (with the attributes *id* and *name* respectively). The name must be used when the identifier is not available, since determining such an identifier may not be trivial.

Finally, a *group* may contain a *location* element, to provide where the group has been declared (see Section 2.6). If the URI of a location in a *group* element is left unspecified (or the entire location element), and if the immediate element containing this *group* element is either a *group* element representing a class or interface, or a *unit* element representing a method, one must assume that the URI of both elements (container and nested) is the same.

The requirements for a *group* element in a DTD are given below:

```
<!ELEMENT group (inheritance*, location?, variable*, (group|unit)*)>
<!ATTLIST group
  id ID #REQUIRED
  type (module|package|class|interface) #REQUIRED
  name CDATA #IMPLIED
  internal CDATA #IMPLIED>

<!ELEMENT inheritance EMPTY>
<!ATTLIST inheritance
  id IDREF #IMPLIED
  name CDATA #IMPLIED>
```

The language of the input program imposes certain additional restrictions; one should assume that these restrictions are also respected in the *group* elements of any SIR (although this is not enforced):

- a *group* element representing a Fortran module may not contain any *group* element;
- a *group* element for C++ may be nested only in a *group* element representing a namespace;
- in a *group* element representing a Java class, the *inheritance* element must always be specified (except if the represented class is *java.lang.Object*);
- the *name* element is never specified in a *group* element representing a Java anonymous class (the *internal* element, however, may be);
- the name of a Java class or interface must *not* be fully specified (that is, it must *not* contain package names), as the full name can be always derived from the SIR structure. In particular, nested classes must not contain the name of the class they are nested within.

2.3 The Elements unit and alias

The *unit* element is used to represent:

- functions, subroutines and the main program in Fortran;
- methods in Java and C++;
- functions in C and C++.

Any *unit* element must specify:

- the type of the unit it represents;
- a unique identifier.

A *unit* element specifies, through the attribute *name*, the name of the unit it represents. The *unit* element may also specify the language of the unit represented (attribute *language*), which is useful when representing C methods linked to Java or Fortran programs. It may also specify an internal, compiler-assigned name for the unit it represents (attribute *internal*). Furthermore, a *unit* element must specify the attribute *instrumentable* with the value *false* (the default is *true*) if the tool that generates the SIR knows that the unit cannot be later instrumented (e.g, it is a library function, but the instrumentation tool can only instrument the source code). Finally, the attribute *virtual* must appear with the value *false* (the default is *true*) if, and only if, one of the following conditions is true:

- the *unit* element represents a Java method declared as private;
- the *unit* element represents a C++ method not declared as virtual.

Nested in a *unit* element there is zero or more *unit* elements, zero or more *group* elements, and zero or more *codeRegion* elements. Similar to *group* elements, a *unit* element may also contain a *location* element, to provide the location where the unit has been declared (see Section 2.6). If the URI of a location in a *unit* element is left unspecified (or the entire location element), and the immediate element containing this *unit* element is either a *group* element representing a class or interface, or a *unit* element representing a method with the same language attribute, one must assume that the URI of both elements (container and nested) is the same.

The declaration and use of variables are represented with the elements *variable* and *variableRef*, described in Section 2.5. When representing a method, function or subroutine, a *unit* element must specify the method (or function, or subroutine) signature by specifying the attribute *arguments* and suplying the identifiers of variables, as described also in Section 2.5.

When representing a Fortran function or subroutine, a *unit* element may specify a name under which the function or subroutine may also be called using the *alias* element.

Note that the fact a function or method is *inline* is ignored.

The syntactic requirements for a *unit* element in a DTD are given below:

```
<!ELEMENT unit (alias?, location?, variable*, arguments*, variableRef*,
                (group|unit|codeRegion)*)>
<!ATTLIST unit
  id ID #REQUIRED
  type (function|subroutine|program|method) #REQUIRED
  name CDATA #IMPLIED
  arguments IDREFS #IMPLIED
  virtual (true|false) #IMPLIED
  internal CDATA #IMPLIED
  language (fortran|java|c|cpp) #IMPLIED
  instrumentable (true|false) #IMPLIED
>
<!ELEMENT alias (#PCDATA)>
```

The language of the input program imposes certain additional restrictions; one should assume that these restrictions are also respected in the *unit* elements of any SIR (although this is not enforced):

- only a *unit* element representing a Fortran function or subroutine may be nested within a *unit* element representing a Fortran subroutine, function, or main program;
- the nesting level for *unit* elements in a *sir* element representing a Fortran program is at most 2;
- for Java and C++ programs, the name used in the *unit* element representing a constructor must be the same name used in the *group* element representing the class where the constructor has been declared;
- for Java programs, the name used in the *unit* element representing a class or interface initializer must be $\<clinit\>$. The "correct" name would be $<$clinit$>$, but the characters $<$ and $>$ may not be used in an element's attribute;
- for Java programs, $\<init\>$ must be the name used in the *unit* element representing an instance initializer;
- a *group* element may be nested within a *unit* element only if the first represents a Java class and the second a Java method (but even if a class is declared inside a method, it may be represented simply nested within the class the method is member of);
- only *unit* elements representing Java classes, Java methods and C functions may be nested within a *group* element representing a Java class;
- a *unit* element may not be nested within another *unit* element if the *sir* element represents a C or C++ program.

2.4 The Elements codeRegion, callee, expression, loopControl, lower, upper, stride, and scheduling

A *codeRegion* element is used to represent a sequence of specific executable program statements and directives in a *unit*. Any *codeRegion* element must specify:

- the type of the program statement it represents in the input program (element *type*);
- a unique identifier (element *id*).

A *codeRegion* element contains zero or more nested *codeRegion* elements and zero or more nested *group* elements (in Java, it is allowed that a class is declared inside a method). It may also contain:

- a *location* element to provide the location of the represented program statement (see Section 2.6);
- a *callee* element, giving the identifier or the name of a method invoked or a function or subroutine called (details under the item *call*, below);
- an *expression* element, giving information about an expression (or expressions) evaluated before the represent code region executes (more details below);
- a *loopControl* element, giving information about the start, stop, and increment expression (or expressions) evaluated by certain kinds of loop constructs (details under the item *loop*, below).
- elements *variable* and *variableRef* to represent the declaration and use of variables (described in Section 2.5).

Because of the complexity and diversity of the different languages and programming models, we do not intend to define a fixed set of allowed types that a *codeRegion* element must follow. In fact, different instrumentors may have their own favorites on what types of code regions are distinguished. However, in the following, we do provide a predefined set of types based on our experiences, which should be regarded only as a recommendation rather than a full specification. In the rest of this document, a *codeRegion* element with a certain type x will be called an x*CodeRegion* element for brevity.

- assignment
  Corresponds to an explicit scalar assignment in the input program, that is, using the operator $=$ and, in the case of Java, C and C++, also the operators $++$, $--$, $+=$, $*=$, and so on (see the type *vector* below for vector assignments). Multiple assignments in a single statement (like $failed = (file = openFile()) == NULL$) should be represented by using nested *expression* elements, but may be also represented by expanding the assignments to several *assignmentCodeRegion* elements (respecting the evaluation order). See an example in Section 3.
- block
  Some language constructs are composed by several blocks. For instance, the *if* construct is composed by the *then*, *else if* (in Fortran), and *else* blocks, and the constructs *switch* and *SELECT* are composed by several blocks to be executed depending on the value of an expression. Instead of creating a new kind of block for each of these constructs, the SIR just defines the generic type *block*, which can be used in any situation. A block can also be used to represent an arbitrary sequence of statements in the input program which cannot be represented by any of the types described in this section.
- if
  Corresponds to the *if* construct in Java, C, C++, and Fortran. An *ifCodeRegion* element has one or more nested *codeRegion* elements, the type of which must be *block*. The *codeRegion* elements inside the first *blockCodeRegion* element corresponds to the *if* part of the *if* construct, and the other *blockCodeRegion* elements, if present, to the *else* or *else if* part. Each *blockCodeRegion* element (except the one corresponding to the "else" part of the *if* construct) may contain also an *expression* element representing the constructs that are evaluated in the respective condition (see an example in Section 3). The expression evaluated in the *if* part of an *if* construct may be also represented as a *codeRegion* element immediately before the *codeRegion* element representing the *if* construct; the use of a nested *expression* element is preferred, though.
  Note: *else if* is allowed only in Fortran; therefore, when representing a Java, C or C++ program, at most two blocks are allowed nested within an *ifCodeRegion* element.

- switch
  Corresponds to the *switch* construct in Java, C or C++, and to the *SELECT* construct in Fortran. The *codeRegion* elements nested within a *switchCodeRegion* element must have the type *block*; each of them corresponds to a ”case” (including the ”default case”) of the *switch* or *SELECT* construct. The presence or absence of an implicit jump after each ”case” must be inferred from the language attribute of the *unit* or *sir* element containing the *switchCodeRegion* element. Moreover, a *switchCodeRegion* element may have one *expression* element representing the condition evaluated by the *switch* or *SELECT* construct (as in *expression* elements for *ifCodeRegion* elements). Although the use of an *expression* element is the preferred way of representing such a condition, it may also be represented by a *codeRegion* immediately before the *codeRegion* representing the *switch* or *SELECT* construct.

- loop
  Corresponds to any kind of loop in the input program: *for*, *while*, *do...while* in Java, C and C++; *DO*, *DO WHILE* (but not *FORALL*) in Fortran. A *loopCodeRegion* element may have either an *expression* element representing the stop condition evaluated by the represented loop construct (as in *expression* elements for *ifCodeRegion* elements) or a *loopControl* element representing the start, stop and increment expressions in these three kinds of loop constructs: *for* (Java, C and C++), *DO* (Fortran). An example is shown in Section 3.

- jump
  Corresponds to an unconditional jump in the input program (*break*, *continue*, and *return* in Java, C and C++, *throw* in Java and C++, *goto* in C and C++, and *GO TO*, *CYCLE*, *EXIT*, and *RETURN* in Fortran). Note that a call to the function *longjmp* is not considered a jump. If the return construct being represented returns a value computed from an expression declared in front of the return statement, this expression should be represented in an *expression* element nested within the *jumpCodeRegion*. Alternatively, it may also be represented as one or more *codeRegion* elements immediately before the *jumpCodeRegion* element representing the *return* construct. The same is valid for the representation of throw constructs, that is, the expression computing the object to be thrown may be represented either as a nested element within the *jumpCodeRegion* or as one or more *codeRegion* elements immediately before it.

- call
  In Fortran, corresponds to a function or subroutine call or to a statement for dynamic storage allocation or deallocation (ALLOCATE, DEALLOCATE, and NULLIFY). In C, it corresponds to a function call, and in C++ to a function call, dynamic storage allocation and deallocation (*new* and *delete*) or a method invocation. In Java, it corresponds to a method invocation or dynamic storage allocation (*new*).
  The creation of class instances, which usually includes dynamic allocation *and* a method (constructor) invocation, must be represented as a single *callCodeRegion*, as if only the constructor were invoked.
  Nested within a *callCodeRegion* element there must be one *callee* element giving either the identifier of the function, subroutine or method invoked (or ”supposed” to be invoked in the case of virtual methods) or, if the identifier is not available, the name of the invoked unit. In C++, when allocating or deallocating memory for a type that cannot be represented as a *unit* in the SIR (like *int*), the callee will be *new* or *delete*, respectively. An *expression* element may also appear nested within a *callCodeRegion* element, indicating assignments and other function calls or method invocations that are performed before the represented call is executed, the results of which will be used as arguments of the call or invocation. Alternatively (but not preferably) these assignments and calls (or invocations) may be represented as *codeRegion* elements appearing immediately before the *codeRegion* representing a call or invocation.
  In indirect calls (for instance, with function pointers), only the signature of the method invoked must be specified for the callee. See an example in Section 3.

- io (Fortran specific)
  Corresponds to an IO statement in Fortran (like PRINT or OPEN). As with *callCodeRegions*, *expression* elements may appear nested within an *ioCodeRegion* to represent other function calls performed before the IO statement is executed.

- try, catch (Java and C++ specific), finally (Java specific)
  Correspond to the construct *try...catch...finally* in Java or *try...catch* in C++.

- where (Fortran specific)
  Corresponds to the *WHERE* construct in Fortran. A *whereCodeRegion* element has one *expression* element, which represents the condition evaluated by the *WHERE* construct (in the same way the *expression* element for *ifCodeRegion*

elements), and one or two nested *blockCodeRegion* elements. The *codeRegion* elements inside the first *blockCodeRegion* correspond to the *where* part in the *WHERE* construct, while the second *blockCodeRegion*, if present, corresponds to the *elsewhere* part.

- forall (Fortran specific) Corresponds to the *FORALL* construct in Fortran. Like a *loopCodeRegion*, a *forallCodeRegion* element may have a *loopControl* element representing the start, stop and increment expressions. It may also contain an *expression* element representing the condition (”scalar mask”) evaluated for each iteration.
- vector (Fortran specific) Corresponds to an explicit vector assignment in the input program. A *vectorCodeRegion* may contain also an *expression* element representing functions called before the assignment take place (for instance, $C = log(A)$).

Furthermore, motivated by OpenMP [4], we also defined a set of *parallel...codeRegions*, which can, in fact, be applied to any similar shared-memory paradigm. Table 1 shows the mapping of OpenMP directives to the corresponding *parallel...codeRegions*.

OpenMP directive	parallel...codeRegion in SIR
PARALLEL	parallelRegion
DO	parallelLoop
SECTIONS	paralleSsections
SINGLE	parallelSingle
WORKSHARE	parallelWorkshare
MASTER	parallelMaster
CRITICAL	parallelCriticalSection
ATOMIC	parallelAtomic
BARRIER	parallelBarrier
FLUSH	parallelFlush
ORDERED	parallelOrdered

Table 1. Mapping from OpenMP directives to *parallel...codeRegions*
.

- parallelRegion
  Corresponds to a code region executed by several threads in parallel.
- parallelLoop
  Corresponds to a work-sharing construct that distributes the iterations of a loop among several threads. The loop is represented by a nested *loopCodeRegion* element. A *scheduling* element may also be nested to inform the scheduling type (static, dynamic, guided, or runtime) and, if applicable, the chunk to be used. Finally, if threads that finish the work they have been assigned do not need to wait until other threads also finish their work, the *nowait* attribute must be specified with the value *true* (default: *false*).
- parallelSections
  Corresponds to a work-sharing construct that distributes the execution of several code regions among several threads. Nested within such a *codeRegion* element there may be only *blockCodeRegions*, each of which representing a code region that is assigned to a thread. If threads that finish the work they have been assigned do not need to wait until other threads also finish their work, the *nowait* attribute must be specified with the value *true* (default: *false*).
- parallelSingle
  Used to group a sequence of code regions that must be executed by only one thread; this sequence is represented by one or more *codeRegion* elements nested within the *parallelSingleCodeRegion*. If the other threads do not need to wait that the thread that executes the code regions finishes its work, the *nowait* attribute must be specified with the value *true* (default: *false*).
- parallelWorkshare
  Corresponds to a work-sharing construct that distributes the execution of several code regions among several threads.

Nested within a *parallelWorkshareCodeRegion* there may be any number of *codeRegion* elements (of any type); the way the execution of the code regions is distributed among threads depends on the library that implements the construct. If threads that finish the work they have been assigned do not need to wait that other threads also finish their work, the *nowait* attribute must be specified with the value *true* (default: *false*).

- parallelMaster
  Used to group a sequence of code regions that must be executed by only one thread, called the master thread; this sequence is represented by one or more *codeRegion* elements nested within the *paralle-masterCodeRegion*.

- parallelCriticalSection
  Used to represent a critical section. Nested within this element there may be any number of *codeRegion* elements. A unique name may be specified (in the attribute *criticalSectionName*) to identify a set of critical sections that must be executed by only one thread at a time. Among the *parallel...CodeRegions*, this is currently the only one that may be used nested within a *sir* element representing a Java program. In Java, however, it is not in general possible, at compile time, to determine a name to give to the *parallelCriticalSectionCodeRegion*, but the expression evaluated to compute the lock to be acquired should be represented either as an *expression* element nested within the *parallelCriticalSectionCodeRegion* (preferred way) or as one or more *codeRegion* elements immediately before the *parallelCriticalSectionCodeRegion* element representing the *synchronized* construct.

- parallelAtomic
  Used to inform that an assignment is performed atomically. Nested within a *parallelAtomicCodeRegion* element there may be only one *codeRegion* element, namely an *assignmentCodeRegion*, which must represent the atomic assignment. Atomicity achieved through library invocations (for instance, using the package java.util.concurrent.atom) must be represented ordinarily with a *callCodeRegion*.

- parallelBarrier
  Corresponds to a language construct that synchronizes all threads within the dynamic scope of a parallel region. Barriers used through library invocations must be represented ordinarily as a *callCodeRegion*.

- parallelFlush
  Corresponds to an explicit construct that provides consistency between a thread (the one that executes the construct) and the main memory.

- parallelOrdered
  Corresponds to a construct that ensures that a sequence of code regions ”is executed in the order in which iterations would be executed in a sequential execution” of a loop [4]. Nested within a *parallelOrderedCodeRegion* there may be any number of *codeRegion* elements (of any type).

As we said, the types of the code region are language and programming model specific and can not be fully specified by the above recommended list. However, the above defined set of types could be the basis for a custom definition.

The requirements for a *codeRegion* element in a DTD are given below. Note that the DTD does not (and cannot) enforce semantic rules involving the *type* attribute of *codeRegion* elements, like the fact that a *callee* element may appear only immediately inside a *callCodeRegion* element.

```
<!ELEMENT codeRegion (callee?, location?,
                      variable*, variableRef*, scheduling?,
                      (expression|loopControl)*, (codeRegion|group)*)>
<!ATTLIST codeRegion
  id ID #REQUIRED
  type CDATA #REQUIRED
  criticalSectionName CDATA #IMPLIED
  noWait (true|false) #IMPLIED
>
<!-- The recommended code region types include
  (block|assign|loop|if|switch|where|jump|call|try|catch|finally|forall|vector|
   parallelRegion|parallelLoop|parallelSections|parallelSingle|
   parallelWorkshare|parallelMaster|parallelCriticalSection|
   parallelAtomic|parallelBarrier|ParallelFlush|ParallelOrdered|
   vector|forall)
-->

<!ELEMENT callee EMPTY>
<!ATTLIST callee
  id IDREF #IMPLIED
  name CDATA #IMPLIED
>

<!ELEMENT expression ((codeRegion|group)+))>

<!ELEMENT loopControl (lower?,upper?,stride?)>

<!ELEMENT lower (codeRegion+)>
<!ELEMENT upper (codeRegion+)>
<!ELEMENT stride (codeRegion+)>

<!ELEMENT scheduling EMPTY>
<!ATTLIST scheduling
  type (static|dynamic|guided|runtime) #REQUIRED
  chunk CDATA #IMPLIED>
```

The order of *codeRegion* elements in the *sir*, as well as the way they are nested, reflect the syntactical order and nesting of the represented program statements in the input program the *sir* represents. For instance, if the program statement (or sequence of program statement) A appears in the input program before the program statement (or sequence of program statements) B, then the *codeRegion* element representing A must appear in the SIR before the *codeRegion* element representing B.

2.4.1 Open Issues Regarding the *unit* Element

- Templates in C++ and Java are not represented
- Overloaded operators in C++ are not represented, although they may represent rather complex functions.
- Extra compiler information
  Sometimes, it is possible to determine through compiler analysis the real method that is going to be invoked (or a set of possible methods). The same is valid for indirect function calls. For instance:

  ```
  if (condition) myfunction = max else myfunction = min;
  x = myfunction(10, 20);
  ```

  or

  ```
  Shape s;
  if (condition) s = new Circle(...) else s = new Square(...);
  s.draw();
  ```

  Even if the compiler has this information, it cannot be represented in the SIR.
- firstprivate, lastprivate, reduction in OpenMP
  It is not clear if they should be represented in the SIR.

2.5 The Elements variable and variableRef

The *variable* element represents the definition of a variable (scalar or array). Each variable must have an attribute of unique *id*, and can have optional attributes like a *name*, a *type*, and *dimensions*. If *dimensions* is defined as -1 or if it is omitted, then the variable is simply a scalar. For arrays, the lower bound and upper bound of each dimension can be specified with one nested element *dimension*, while the *index* attribute indicates which dimension is being described. The type used for a *variable* element is language dependent (that is, this specification does not dictate the name under which the type of an variable must be represented), but it should be used consistently throughout the input program representation.

As usual, the *location* element informs where a variable is declared in the input program.

The DTD segment for *variable* element is given below:

```
<!ELEMENT variable (location?, dimension*)>
<!ATTLIST variable
  id ID #REQUIRED
  name CDATA #IMPLIED
  type CDATA #IMPLIED
  dimensions CDATA #IMPLIED
>
<!ELEMENT dimension EMPTY>
<!ATTLIST dimension
  index CDATA #REQUIRED
  upperBound CDATA #REQUIRED
  lowerBound CDATA #REQUIRED
>
```

As method, function and subroutine arguments are in fact variables, they are also represented as such; in addition, the attribute *arguments* of a unit will contain a list of identifiers referring to the variables that are arguments in the unit.

References to variables in each unit and code region are represented by *variableRef* elements. Each *variableRef* element must specify *targetId*, which is used to identify the variable that it references. The optional attribute *accessType* can also be supplied to indicated if the variable is read, written, or both. The DTD segment for *variableRef* element is given as follows:

```
<!ELEMENT variableRef EMPTY>
<!ATTLIST variableRef
    targetId IDREF #REQUIRED
    accessType ( read | write | readwrite ) #IMPLIED
>
```

2.6 The location Element

A *location* element represents the location of a unit, a program statement, a variable declaration, a directive or a sequence of program statements and directives in a file. The *location* element contains attributes for representing the start line, the start column, the end line and the end column the represented code occupies in a ”file” (not necessarily all of them need to appear in the element). The location of a file is given by the attribute *uri*, and does not need, in fact, to refer to a file, but to any resource. If the resource where the represented code is defined is not the same as the resource a nested unit, program statement or directive is defined, the *location* element in the nested unit or program statement must also be specified.

The requirements for a *location* element in a DTD are given below:

```
<!ELEMENT location EMPTY>
<!ATTLIST location
  startLine CDATA #IMPLIED
  startColumn CDATA #IMPLIED
  endLine CDATA #IMPLIED
  endColumn CDATA #IMPLIED
  uri CDATA #IMPLIED>
```

An example that uses the *location* element is shown in Section 3. The syntax of a *uri* attribute can be found in [5].

3 Examples

Figure 1 shows two ways of representing several assignments appearing in a single statement, as well as how variables are represented.

```
int failed;
FILE* f;
failed = (f = fopen("file.txt", "r+")) != NULL;
```

(a) C code

```
<unit type="function" name="fopen" arguments= "v1 v2"
      instrumentable="false" id="u1">
  <variable type="char*" id="v1"/>
  <variable type="char*" id="v2"/>
</unit>
...
<variable type="integer" name="failed" id="v3"/>
<variable type="FILE*" name="f" id="v4"/>
<codeRegion type="assignment" id="a1">     <!-- failed = ... -->
  <variableRef targetId="v3" accessType="write"/>
  <variableRef targetId="v4" accessType="read"/>
  <expression>
    <codeRegion type="assignment" id="a2"> <!-- f = ... -->
      <variableRef targetId="v4" accessType="write"/>
      <expression>
        <codeRegion type="call" id="c1">   <!-- fopen() -->
          <callee id="u1"/>
        </codeRegion>
      </expression>
    </codeRegion>
  </expression>
</codeRegion>
```

(b) SIR mapping using the element *expression*

```
<codeRegion type="call" id="c1">        <!-- fopen() -->
  <callee id="u1"/>
</codeRegion>
<codeRegion type="assignment" id="a1"> <!-- f = ... -->
  <variableRef targetId="v4" accessType="write"/>
</codeRegion>
<codeRegion type="assignment" id="a2"> <!-- failed = ... -->
  <variableRef targetId="v3" accessType="write"/>
  <variableRef targetId="v4" accessType="read"/>
</codeRegion>
```

(c) SIR mapping without the element *expression*

Figure 1. Two ways of mapping multiple assignments to SIR

Figure 2 illustrates the mapping of inheritance and constructors of Java classes, as well as method (in this case, constructor) invocations.

```
package example;

class MyClass extends java.awt.Button
              implements Runnable, java.awt.event.KeyListener {
    MyClass(String s) { super(s); }
    ...
}
```

(a) Java code

```
<group type="package" name="java" id="p1">
  <group type="package" name="lang" id="p2">
    <group type="interface" name="Runnable" id="i1"/>
  </group>
  <group type="package" name="awt" id="p3" instrumentable="false">
    <group type="class" name="Button" id="c1">
      <unit type="method" name="Button" arguments="v1" id="m1">
        <variable type="java.lang.String" id="v1"/>
      </unit>
    </group>
    <group type="package" name="event" id="p4">
      <group type="interface" name="KeyListener" id="i1"/>
    </group>
  </group>
</group>
<group type="package" name="example" id="p5">
  <group type="class" name="MyClass" id="c2">
    <unit type="method" name="MyClass" id="m2">
      <codeRegion type="call" id="cr1">
        <callee id="m1">
      </codeRegion>
    </unit>
    ...
  </group>
</group>
```

(b) SIR mapping

Figure 2. How inheritance and constructor invocation are mapped

Figure 3 illustrates the mapping of an *if* construct in C to an *ifCodeRegion*, including the use of the elements *blockCodeRegion*, *callee*, and *expression*.

```
if (f(n) > g(m)) {
    a = 10;
} else {
    flag = false;
}
```

(a) C code

```
<!-- assume that the id of f is "f" and the id of g is "g" -->
<codeRegion type="if" id="i1">                   <!-- if (...) -->
  <codeRegion type="block" id="i2">
    <expression>
      <codeRegion type="call" id="i3">           <!-- f(n) -->
        <callee id="f"/>
      </codeRegion>
      <codeRegion type="call" id="i4">           <!-- g(m) -->
        <callee id="g"/>
      </codeRegion>
    </expression>
  <codeRegion type="assignment" id="i5"/>       <!-- a = 10 -->
</codeRegion>
<codeRegion type="block" id="i6">                <!-- else -->
  <codeRegion type="assignment" id="i7"/>        <!-- flag = false -->
</codeRegion>
</codeRegion>
```

(b) SIR mapping

Figure 3. How if constructs and function calls are mapped

Figure 4 illustrates the mapping of a FORALL loop in Fortran to a *loopCodeRegion*, including the use of the *loopControl* element.

```
FORALL (i = fg(5):gh(100):hi(2), j = 4:mn(8))
    ...
END FORALL
```

(a) Fortran code

```
<!-- assume that the functions have identical ids and names -->
<codeRegion type="loop" id="i1">
  <loopControl>  <!-- i -->
    <lower>  <!-- fg(5) -->
      <codeRegion type="call" id="i2">
        <callee id="fg"/>
      </codeRegion>
    </lower>
    <upper>  <!-- gh(100) -->
      <codeRegion type="call" id="i3">
        <callee id="gh">
      </codeRegion>
    </upper>
    <stride>  <!-- hi(2) -->
      <codeRegion type="call" id="i4">
        <callee id="hi"/>
      </codeRegion>
    </stride>
  </loopControl>
  <loopControl>  <!-- j -->
    <upper>  <!-- mn(8) -->
      <codeRegion type="call" id="i5">
        <callee id="mn"/>
      </codeRegion>
    </upper>
  </loopControl>
  ...
</codeRegion>
```

(b) SIR mapping

Figure 4. How FORALL is mapped

Figure 5 shows the mapping of pointer functions in C.

```
void sort(void *array, int size, int (*cmpfunc)(const void *, const void *)) {
    ...
    cmpfunc(a, b);
    ...
}
```

(a) C code

```
<unit type="function" name="sort" arguments="v1 v2 v3" id="f1">
  <variable name="array" type="void*" id="v1"/>
  <variable name="size" type="int" id="v2"/>
  <variable name="cmpfunc" type="(int)(const void *, const void *)" id="v3"/>
  ...
  <codeRegion type="call" id="c2">
      <callee id="u1"/>
  </codeRegion>
  ...
</unit>
```

(b) SIR mapping

Figure 5. How pointer functions are mapped

Figure 6 shows the mapping of overloaded functions in Fortran.

```
INTERFACE PHI
    FUNCTION IPHI(X)
        INTEGER IPHI, X
    END FUNCTION IPHI

    FUNCTION RPHI(X)
        REAL RPHI, X
    END FUNCTION RPHI
END INTERFACE PHI
! function contents not important
```

(a) Fortran code

```
<unit type="function" name="IPHI" arguments="v1" id="f1">
  <variable name="X" type="INTEGER" id="v1"/>
  <alias>PHI</alias>
  ...
</unit>
<unit type="function" name="RPHI" arguments="v2" id="f2">
  <variable name="X" type="REAL" id="v2"/>
  <alias>PHI</alias>
  ...
</unit>
```

(b) SIR mapping

Figure 6. How overloaded functions in Fortran are mapped

Finally, figure 7 shows a piece of Fortran code mapped to a SIR including the *location* element, and also how an IO statement is mapped to an element in the SIR.

file F1.f90

```
        column 123456789012345678901234
    line 1:         SUBROUTINE f(x)
    line 2:             REAL :: x
    line 3:             INCLUDE "F2.f90"
    1ine 4:         END SUBROUTINE f
```

file F2.f90

```
        column 12345678901234567890
    line 1:         PRINT *, foo(1)
```

(a) Fortran code

```
<!-- assume that the id of PRINT is "print" and the id of foo is "foo"-->
<unit type="subroutine" name="f" id="i1">
  <location startLine="1" startColumn="5" endLine="4" endColumn="20"
            uri="file:///home/joe/programs/F1.f90"/>
  <codeRegion type="io" id="i2">
    <location startLine="1" startColumn="5" endLine="1" endColumn="19"
              uri="file:///home/joe/programs/F2.f90"/>
    <expression>
      <codeRegion type="call" id="i3">
        <location startLine="1" startColumn="14" endLine="1" endColumn="19"/>
        <callee id="foo"/>
      </codeRegion>
    </expression>
    <callee id="print"/>
  </codeRegion>
</unit>
```

(b) SIR mapping

Figure 7. How IO statements in Fortran are mappend, including location information

4 DTD to XML Schema Translation

An *XML schema* is itself an XML document that describes the structure and constrains the contents of XML documents by following the *XML schema language specification* [2]. It substantially reconstructs and considerably extends the capabilities found in XML DTDs (but does not allow to define semantic rules either). The corresponding XML schema for the DTD shown in Section 1.1 could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="employee">
    <xs:complexType>
      <xs:attribute name="matr" type="xs:ID" use="required"/>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="marriedTo" type="xs:IDREF"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="staff">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element ref="employee"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Rules for translating DTDs to XML schemas can be found in [3]; in addition, the type restrictions shown in Table 2 should be used.

Element	Attribute	Type in the XML Schema
location	startLine	nonNegativeInteger
	startColumn	nonNegativeInteger
	endLine	nonNegativeInteger
	endColumn	nonNegativeInteger
	url	anyURI
scheduling	chunk	positiveInteger
variable	dimensions	nonNegativeInteger
dimension	index	positiveInteger
dimension	lowerBound	integer
dimension	upperBound	integer

Table 2. Types to be used when converting to XML Schema the elements and attributes of the DTD describing the SIR grammar

.

5 Conclusion

This document has shown how to represent programs in several languages using a neutral format defined in XML; this approach not only reduces the dependence of performance tools on a specific instrumentation engine, but also increases their portability, making it possible to support new languages and instrumentations tools at low cost.

Nevertheless, a compromise was sometimes necessary in order to unify under a single SIR element several constructs that fundamentally represent the same idea. For example, a C++ programmer may find strange that a namespace is called a ”package”, and an object-oriented purist might complain that a *call* element is used to represent a method invocation. Another problem is that not always a lowest common denominator can be found; some concepts are specific for only one language or paradigm and do not have a parallel in other languages.

We must also note that not everything that *can* be represented with SIR *must* be represented. For example, when generating the SIR from a binary, only little information will be available except for the program structure and the function calls. The SIR in this case will be extremely reduced, but it will still be valid.

Admittedly, much potentially useful information for performance analysis is not covered by this representation. We believe, however, that the benefits of having a common format accepted by several instrumentation engines justify these omissions: with little or no effort, performance tools could change the target language or platform, or even be extended to analyse and compare programs running in heterogeneous environments.

References

[1] Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/REC-xml.

[2] XML Schema Part 1: Structures. http://www.w3.org/TR/xmlschema-1.

[3] A Conversion Tool from DTD to XML Schema. http://www.w3.org/2000/04/schema_hack.

[4] OpenMP Fortran Application Program Interface Version 2.0.
http://www.openmp.org/specs/mp-documents/fspec20.pdf.

[5] RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt.